Chapter 3

The dbx Debugger

The dbx debugger is a source-level, symbolic debugger. As a source-level debugger, it allows you to control the execution of individual source lines in a program and to set stops, or breakpoints, at specific source lines. As a symbolic debugger, it also allows you to refer to program locations by their symbolic names.

The dbx debugger supports multiple programming languages, and can evaluate and display values from different languages. In addition, it provides symbol lookup according to the scoping rules of a specific language.

The dbx debugger operates on an executing program and controls the execution according to your specifications. It allows you to follow the execution of your program interactively and enables you to examine or alter the state of your program at any point. It also allows you to access your source file for display and editing purposes.

Before program execution starts, dbx allows you to enter debugger commands (such as setting a breakpoint in your program). You can then enter the debugger's **run** command, which creates a user process and starts program execution. Your program executes until a breakpoint or an exception condition occurs, at which point dbx again gains control and prompts for input. (dbx runs interactively as a separate process; that is, it is not associated with the user process for the executing program.)

NOTE

You should compile VAX C programs that require debugging without source code optimizations. The VAX C compiler code optimizations will have an unpredictable effect on the debugging environment. When you use the **–g** option on the **vcc** command line optimization is not performed.

Sections in this chapter address the following topics:

- The **dbx** command line used to invoke the dbx debugger
- The conventions observed by dbx
- The effect of compiler optimizations on the dbx debugger
- The commands available within the dbx debugger
- An example of a debugging session

3.1 Invoking the dbx Debugger

The command line that invokes the dbx debugger has the following form:

dbx [–c *file*] [–i] [–I *dir*] [–k] [–r] [*objfile*] [*coredump*]

–c *file*
Executes the **dbx** commands in the file before reading from standard input.

–i
Forces dbx to act as if the standard input device is a terminal.

–I *dir*
Adds *dir* to the list of directories that are searched when looking for a source file. Normally, dbx looks for source files in the current directory and in the directory where the file that is being debugged is located. You can set the directory search path with the **use** command. (Section 3.4 contains more information on the **use** command.)

–k
Maps memory addresses. This facility is useful when you are debugging functions within the kernel.

–r
Executes objfile immediately. If it terminates successfully, dbx exits. Otherwise, the reason for termination is reported and dbx does not exit. When the **–r** option is specified and the standard input device is not a terminal, dbx reads from /dev/tty.

If the **–r** option is not specified, dbx issues a prompt and waits for a command.

objfile
Specifies an executable file produced by the **vcc** command. If you do not specify an output file name on the **vcc** command line, the file is given the name a.out by default.

You must specify the **–g** option on the **vcc** command line to produce the symbol information needed by dbx in objfile. The file contains a symbol table that includes the name of all source files translated during the compilation process. You can access these source files while you are using the debugger.

coredump
If the file core exists in the directory, or you specify a coredump file, you can use dbx to examine the state of the program at the time a fault occurs.

NOTE

Functions compiled without the **–g** option are stepped over by dbx, unless you explicitly set a breakpoint in the function. However, even if you set a breakpoint, you will not be able to fully analyze what is happening in the function because information about the symbols in program components compiled without the **–g** option—except for global symbols—is not available to dbx.

3.2 dbx Conventions

Understanding the conventions discussed in the following sections will assist you when using the dbx debugger.

3.2.1 dbx Initialization Files

You can build an initialization file containing dbx commands that you want to have in effect when you begin debugging sessions. When you enter the **dbx** command, the debugger first searches the current directory for an initialization file. If it fails to find one in the current directory, it searches your home directory. When the debugger finds an initialization file, it executes the dbx commands contained in the file.

In searching for the initialization file, the debugger bases its search on the combination of init and up to the first eight characters of the debugger's name. The debugger normally looks for .dbxinit. If you rename the debugger, it searches for an initialization file consisting of the first eight characters of the new name with the init suffix. If, for example, you rename dbx as ABCDEFGHI, the debugger searches for an initialization file called .ABCDEFGHinit.

3.2.2 Command Line Retention

Each time you enter a **dbx** command, dbx saves that command line until you enter another command. If you want to repeat the execution of the previous **dbx** command line, press the RETURN key.

This feature is useful for repeating a debugging operation (for example, stepping through a program, one instruction at a time, using a **next** or **step** command). You can also use command-line retention to examine consecutive memory locations. The current memory pointer is not automatically updated. The following command sequence allows you to step through memory. This example uses the machine-level debugging commands described in Section 3.4.2.

```
(dbx)  0x100/i
***dbx display***
(dbx)  ./i                    <== updates current location pointer
***dbx display***
(dbx)  RETURN
***dbx display***
(dbx)  RETURN
         .
         .
         .
```

3.2.3 Expressions in dbx Commands

You can enter numeric expressions during your dbx debugging sessions. Expressions in dbx commands follow the syntax of the C language. Table 3–1 lists the dbx operators.

Table 3–1: dbx Operators

dbx Operator	Operation
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
+	Addition
–	Subtraction
*	Multiplication
/	Division

As in C, you must place array subscripts within square brackets ([ ]). Consider the following example:

```
(dbx)  assign array1[1]=1
```

3.3 Debugging Optimized Programs

The VAX C compiler performs code optimizations by default. Unlike many other optimizing compilers, VAX C does not require additional compile time when optimizing. For many applications, compile and link time is increased when optimization is not used because of the resulting increase in the size of the object program.

Use the **vcc** command line option **–V nooptimize** if you plan to debug VAX C programs with the dbx debugger because code optimizations may affect the debugging environment. To encourage the use of this option, the **–g vcc** command line option turns optimization off.

Debugging optimized code is recommended only under special circumstances. For example, if a problem disappears when optimization is not selected you must try to debug optimized code.

One aid to debugging optimized code is the **–V machine_code=interspersed** option. This option allows you to generate a listing file that shows the compiled code produced for each source line in your program. By referring to a listing of the generated code, you can see exactly how the compiler optimizations affected your code. This helps you to determine the debugging commands needed to isolate the problem.

Another aid is a set of messages that dbx issues when you try to perform a dbx operation on a language construct that has been optimized. In these instances, one of the following optimizations has occurred:

- Optimized variables. If the VAX C compiler determines that a memory location for a variable is not needed for the correct operation of a program, the compiler informs dbx that the variable exists but that no memory is allocated to it. In this case, dbx prints the following message:

  ```
  symbolic information not available for variable variable
  ```

When you receive such a message, you must either find another way to obtain the information you need (perhaps by examining the machine code listing), or you must recompile without specifying the **–V optimize** option on the **vcc** command line.

- Optimized lines. If the VAX C compiler determines that an entire statement is not needed for correct operation of the program, that statement is not represented in the object code. As a result, dbx cannot use such statements to set stops (breakpoints) or tracepoints. If you try to set a stop on an optimized line, dbx prints the following message:

  `no executable code at line` *`line-number`*

  If you encounter this situation, you can usually set the stop or trace on an adjacent line that has not been optimized.

  You will also receive the preceding message if you try to set a stop at either a line that contains only data declarations or at a comment line. This occurs whether or not optimization is in effect.

3.4 dbx Commands

Table 3–2 provides a summary of dbx commands. The commands fall into the following functional categories:

- General-purpose commands
- Execution and tracing commands
- Scope and variable handling commands
- Source-file access commands
- Machine-level debugging commands

Sections 3.4.1 and 3.4.2 describe these commands in greater detail.

Table 3–2: dbx Command Summary

Command Category	Command	Description
General-Purpose Commands	**alias**	Establishes an alias for an established dbx command name or lists current aliases.
	help	Displays a summary of dbx commands.
	quit	Terminates dbx processing.
	sh	Passes a command line to the default shell for execution.
	unalias	Removes the **alias** associated with a name.
Execution and Tracing Commands	**call**	Executes the specified function.
	catch	Traps a specified signal condition in the debugger. (In this case, your program will not receive the signal unless you enter a **cont** command.)

(continued on next page)

Table 3–2 (Cont.): dbx Command Summary

Command Category	Command	Description
	cont	Continues execution from the point of suspension. (The **cont** command is also used to continue signal processing.)
	delete	Removes specified traces and stops.
	ignore	Stops trapping a specified signal condition. (In this case, the debugger will automatically pass the signal to your program.)
	next	Executes the current source statement—executing any called subprograms—and stops at the next source line.
	rerun	Reruns a program using the arguments specified on a previous **run** command.
	return	Stops dbx processing at the next executable instruction following a return from the current subprogram or from a specified subprogram.
	run	Begins execution of a program.
	source	Executes dbx commands contained in a given file.
	status	Displays the traces and stops that are in effect.
	step	Executes one source line—stopping at the first line of a called subprogram.
	stop	Suspends execution when a line is reached, when a function is called, or when a condition is met.
	trace	Traces execution of a line, traces calls to a function, or traces changes to a variable. Also, displays the results of specified expressions when a given line is reached.
Scope and Variable Handling Commands	**assign**	Assigns the value of an expression to a variable.
	down	Changes the current scope to a lower stack count level.
	dump	Displays the names and values of all active variables.
	func	Displays or changes the current scope.
	print	Displays the value of an expression.
	set	Assigns values to debugger variables.
	unset	Deletes the debugger variable associated with the name.
	up	Changes the current scope to a higher stack count level.
	whatis	Displays the declaration of a name.
	where	Displays the currently active functions.
	whereis	Displays the full qualification of all occurrences of a symbol.
	which	Displays, in the current scope, the full qualification of a symbol.

(continued on next page)

Table 3–2 (Cont.): dbx Command Summary

Command Category	Command	Description
Source-File Access Commands	**edit**	Begins an editing session on a specified file using the default editor (vi).
	file	Displays or changes the current source file.
	list	Displays a range of source lines or a specified function.
	use	Establishes a directory search path.
	/ **?**	Searches forward or backward in the current source file for the specified pattern.
Machine-Level Debugging Commands	*address*	Displays the contents of memory locations at the specified address (or within the specified address range).
	nexti	Executes the current machine instruction and stops at the next machine instruction.
	stepi	Executes the current machine instruction, and stops at the first machine instruction of a called function.
	stopi	Suspends execution when an address is reached, a condition is met, or a function is called.
	tracei	Traces execution of an address.

3.4.1 Source-Level Debugging Commands

This section provides expanded descriptions of the dbx commands. Section 3.4.2 describes the machine-level commands.

alias *newcommandname oldcommandname* alias *commandname*
alias *commandname "string"*
alias *commandname(parameter) "string"*
alias
Responds to newcommandname as though it were oldcommandname. You can use either name to enter the command.

If you enter the **alias** command with a name, a string, and an optional parameter, **alias** executes the string each time you enter the name. For example, to define an alias called b that sets a stop at a particular line, issue the following dbx command:

```
dbx  alias b(x) "stop at x"
```

When you enter the command b(12), dbx stops execution at line 12.

If you enter only one command name, dbx displays the aliases for that name.

If you enter the **alias** command with no arguments, dbx displays all the aliases that you have established.

assign *variable* = *expression*
Assigns the value of the expression to the variable. The value and the variable must be of the same data type.

You cannot assign values to registers.

call *function(parameters)*
Executes the named function. The dbx debugger passes all parameters by reference; that is, it passes the address of the parameter arguments.

catch *number*
catch *name*
Traps the signal, specified by its name or number, in dbx rather than passing it to the executing program. The **catch** command remains in effect until it is terminated by the **ignore** command.

This command is useful when you are debugging programs that handle signals, such as interrupts. (See the *Guide to Languages and Programming* for more information about signals and signal handling.) By default, dbx traps all signals except SIGCONT, SIGCHILD, SIGALRM, and SIGKILL. The dbx debugger does not keep track of a signal number after it traps it. If dbx traps a signal and you want to pass it to a signal handler routine, you must enter a **cont** command specifying the appropriate signal name or number. (This section describes the **cont** and **ignore** commands.)

cont [*number*]
cont [*name*]
Continues execution from the point at which the process stopped. Execution cannot be continued if the process has finished (that is, it has called the standard exit procedure) or if it has not started executing. dbx does not allow the process to exit so you cannot examine the program state after execution is completed.

The number is an integer (1 through 32) that represents a signal. When you specify the number or name of a signal, you instruct the program that receives that signal to continue processing as if the signal was received. (See the description of the **catch** command that appears in this section.) For example, the following command specifies that a program that receives signal 4 is to continue and that control is to be passed to the appropriate signal handler:

```
(dbx)  cont 4
```

delete *commandnumber, ..., commandnumber*
delete *
Removes the traces or stops corresponding to the given command numbers. (The numbers associated with traces and stops are displayed by the **status** command.)

The **delete *** command removes all existing breakpoints and tracepoints.

down [*count*]
Moves the current scope, which is used for resolving names, down the stack by the number of levels specified in count. The default count is 1. See the **up** and **func** commands for more information.

dump [> *filename*]
Displays the names and values of all active variables in the specified procedure or the current procedure. If you use a period (.) in place of the function name, all active variables are dumped. Active variables are variables in common blocks or in functions that are active. See the **print** command for a list of the supported data types.

edit [*filename*]
edit *functionname*
Invokes the default editor (vi) and accesses the file filename or, if none is specified, the current source file. If you specify a function name, the editor accesses the file containing the specified function.

file [*filename*]
Changes the current source file to filename. If you omit the filename, dbx displays the name of the current source file.

func [*function*]
Changes the current function. If you omit the function, dbx displays the name of the current function. Changing the current function implicitly changes the current source file to the one that contains the function; it also changes the current scope used for name resolution.

The **func** command is similar to the **up** and **down** commands. The major difference is that **up** and **down** change scope based on stack frame counts and **func** changes scope based on the function name that you specify. You can use **func** to change the scope (the current function) to an inactive function (that is, a function not on the stack). However, the **up** and **down** functions are only effective with active functions.

help
Displays a summary of dbx commands.

ignore *number*
ignore *name*
Stops trapping the signal, specified by its name or number and passes it directly to the user command program **ignore**. The **ignore** command disables the **catch** command. (A description of the **catch** command appears in this section.)

The **ignore** command is useful when you are debugging programs that handle signals, such as interrupts. By default, dbx traps all signals except SIGCONT, SIGCHILD, SIGALRM, and SIGKILL. The dbx debugger does not keep track of a signal number after it traps it. If dbx traps a signal and you want to pass it to a signal handler routine, you must issue a **cont** command specifying the appropriate signal number. (A description of the **cont** command appears earlier in this section.)

You can disable trapping for a particular signal using the **ignore** command, as follows:

```
(dbx) ignore 4
```

Signal number 4 (SIGILL) applies to interrupts caused by illegal instructions. If an illegal instruction occurs after you issue the preceding **ignore** command, dbx ignores the signal and gives the program a chance to handle the signal. (See the *Guide to Languages and Programming* for additional information about signals and signal handling.)

list [*linenumber*[,*linenumber*]]
list *function*
Lists the lines in the current source file from the first line number specified to the second line number specified, inclusive. If you omit line numbers the next ten lines are displayed. If you specify the name of a subprogram, ten lines (five above and five below the first statement in the subprogram) are displayed.

The second form of the **list** command, **list** *function*, has the same effect as the **file** command; that is, it changes the current source file.

next
Executes up to the next source line. The **next** command is different from the **step** command. If the current line contains a call to a function and you enter the **next** command, execution continues until the next source line; that is, execution does not stop within the called function. In contrast, if you enter the **step** command execution stops at the first line of the called function.

print *expression*[,*expression* ...]
Displays the values of the specified expressions. Array expressions are always subscripted by brackets ([ ]). You can reference variables that have the same identifier as one within the current scope, as functionname.variable. You can use the backslash operator (\ ) in the construct expression\typename to display the results of an expression in the format of a given type.

The **print** and **dump** commands display variables with the following data types:

Data Type	Size
int **long** **long int**	32 bits
unsigned **unsigned int**	32 bits
short **short int**	16 bits
unsigned short	16 bits
char **unsigned char**	8 bits
float	32 bits
double	64 bits

quit

Terminates the debugging session.

rerun [*args*][< *filename*] [> *filename*]
Starts executing the program specified with filename by passing args as command-line arguments. If, for example, argc and argv are arguments expected by a C program, you can include them on the **rerun** command line. Use left and right angle brackets (<>) to redirect input or output in the usual manner.

If you specify arguments on the **rerun** command, dbx appends them to the original argument list specified by the **run** command. If you omit arguments from the **rerun** command, dbx passes the previous argument list to the program. Otherwise, the **rerun** command is identical to the **run** command.

If the object file associated with filename has been written since the last time the symbolic information was read in, dbx reads in the new information.

return [*function*]
Continues until a return to function is executed, or until the current subroutine or function returns if function is omitted.

run [*args*] [< *filename*] [> *filename*]
Starts executing the program specified with filename by passing args as command line arguments. If, for example, argc and argv are arguments expected by a C program, you can include them on the **run** command line. Use left and right angle brackets (<>) to redirect input or output in the usual manner.

If you specify arguments on the **rerun** command, they are appended to the original argument list specified by the **run** command. If you omit arguments from the **rerun** command, the previous argument list is passed to the program. Otherwise, the **rerun** command is identical to the **run** command.

If the object file associated with filename has been written since the last time the symbolic information was read in, dbx reads in the new information.

set *variable* = *expression*
Assigns values to debugger variables. The names of these variables cannot conflict with variable names in the program being debugged. The following variables have a special meaning within dbx:

$frame	You can set this variable to an address. The dbx debugger uses the stack frame pointed to by the address to perform stack traces and access local variables.
$hexchars **$hexints** **$hexoffsets** **$hexstrings**	When one of the variables is set, dbx prints out the hexadecimal value for characters, integers, offsets, or strings.
$listwindow	You can set this variable to a numeric value that will be used by the **list** command. The value specifies the number of lines to list around a function, or the number of lines to display when the **list** command is entered without a parameter. If you use the **set** command to set this variable without specifying a numeric value, dbx uses the default value of 10.
$mapaddrs	When you set this variable, dbx starts mapping addresses and continues to map addresses until you revoke the variable setting.
$unsafecall **$unsafeassign**	When you set these variables, strict type checking is omitted within specific situations. The **$unsafecall** variable omits strict type checking during subroutine and function calls. The **$unsafeassign** variable omits strict type checking for the two sides of an assignment statement.

sh *commandline*
Passes the command line to the shell for execution. The mechanism used to return to dbx varies according to the shell being used. For example, for the Bourne shell, you issue CTRL/D. Your shell is determined when you log in.

source *filename*
Executes the dbx commands contained in the specified file.

status [> *filename*]
Displays the currently active **trace** and **stop** commands. You can ignore any gaps in the sequence of numbers shown by the **status** command; the gaps occur because dbx uses some of the numbers internally.

step
Executes up to the next line. The **step** command is different from the **next** command. The **step** command proceeds to the next line you can execute. As a result, if the current line contains a function call, execution stops on the first line

of the called function. However, with the **next** command, execution continues until the next source line so it does not stop within the called function.

stop if ***condition***
stop at ***sourcelinenumber*****[if** ***condition*****]**
stop in ***function*** **[if** ***condition*****]**
stop ***variable*** **[if** ***condition*****]**
Stops execution when one of the following conditions apply: the given condition is true, the given line is reached, the given subroutine or function is called, or the given variable is modified. The description of the **trace** command describes how to specify **stop** command arguments.

trace[in ***function*****][if** ***condition*****]**
trace ***sourcelinenumber*** **[if** ***condition*****]**
trace ***function*****[in** ***function*****][if** ***condition*****]**
trace ***expression*** **at** ***sourcelinenumber*** **[if** ***condition*****]**
trace ***variable*** **[in** ***function*****] [if** ***condition*****]**
Displays tracing information when the program executes. A number is associated with each **trace** command. You must reference this number when using the **delete** command to turn off tracing.

If you specify the **in** clause with a function, tracing is in effect only within the given subroutine or function.

The variable condition is a Boolean expression that is evaluated prior to displaying the tracing information. If the result is false, the information is not displayed. Section 3.2.3 describes the operators used in conditional expressions.

The first argument—sourcelinenumber, function, expression, or variable—specifies what is to be traced as follows:

- If the argument is a source line number, the line is displayed immediately prior to execution. To specify source line numbers in a file other than the current file, precede the line numbers with the name of the file in quotes followed by a colon (:). For example, "xyz.c":17 specifies line number 17 in the file xyz.c.
- If the argument is a subroutine or function name, information is displayed every time the subroutine or function is called, telling what function called it, from what source line it was called, and what parameters were passed to it. In addition, a message noting the return is displayed and, if it is a function return, the value being returned is also displayed.
- If the argument is an expression with an **at** clause, the value of the expression is displayed each time the identified source line is reached.
- If the argument is a variable, the name and value of the variable is displayed if there is any change to the value or the name of the variable. The previous value of the variable, prior to the change, is also displayed. Execution is much slower using this form of tracing.
- If the argument is omitted, all source lines are displayed before they are executed. Execution is much slower during this form of tracing.

unalias ***name***
Removes the **alias** associated with name.

unset ***name***
Deletes the debugger variable associated with the name.

up [*count*]
Moves the current scope, which is used for resolving names, up the stack by the number of levels specified in count. The default value of count is 1. See the **down** and **func** commands.

use *directory-list*
Sets the list of directories to be searched when looking for source files. You can also do this by using the **–I** option on your **dbx** command line.

whatis *name*
Displays the declaration of the given name. You can qualify the name using the function names described under the **print** command in this section.

where
Displays a list of the active functions.

whereis *symbol*
Displays the full qualification of the specified symbol for each occurrence of the symbol in all functions in the program. The order in which the symbols are displayed is not meaningful.

which *symbol*
Displays the full qualification of the given symbol within the currently active function in the stack.

/expression/
Searches forward in the current source file for the specified expression.

?expression?
Searches backward in the current source file for the specified expression.

3.4.2 Machine-Level Debugging Commands

Machine-level commands allow you to examine instructions that result from the expansion of a VAX C statement.

This section provides expanded descriptions of machine-level commands in alphabetical order.

***address, address* / [*mode*]**
[*address*] / [*count*] [*mode*]
***symbol* / *count* [*mode*]**
Displays the contents of memory, starting at the first address and continuing up to the second address or until count items are displayed (the default count is 1). The address is a memory location expressed as a hexadecimal, decimal, or octal number. Note that registers 0-15 are denoted by $rn (n is the number of the register) or, for registers 12 through 15, by $ap, $fp, $sp, or $pc, respectively.

Symbolic addresses are specified by preceding the name with an ampersand (&). Addresses can be expressions made up of other addresses and the plus (+), minus (–), and indirection (unary *) operators.

A symbolic name (symbol) that has been assigned the value of a memory location can also be specified, with enclosing parentheses, to achieve the same effect as address. For example, if the pc contains the address 0x100, then the following two commands would both print 10 instructions starting at 0x100:

```
(dbx)  0x100/10 i
(dbx)  ($pc)/10 i
```

If you replace the address with a period (.), the address following the one most recently displayed is used. The mode option specifies how memory is to be displayed; if mode is omitted, the previous mode specified is used. You must include the slash (/) even if you do not specify count or mode. The default mode is X. The following modes are supported:

i	Displays the machine instruction
d	Displays a short word (16 bits) in decimal
D	Displays a long word (32 bits) in decimal
o	Displays a short word in octal
O	Displays a long word in octal
x	Displays a short word in hexadecimal
X	Displays a long word in hexadecimal
b	Displays a byte in octal
c	Displays a byte as a character
s	Displays a string of characters terminated by a null byte
f	Displays a single-precision real number, *float*
g	Displays a double-precision real number, *double*

nexti
Executes up to the next machine instruction. The **nexti** command is different from the **stepi** command. The **stepi** command proceeds to the next instruction. As a result, if the current instruction contains a function call, execution stops on the first instruction of the called function. In contrast, if you enter the **nexti** command, execution does not stop within the called function.

stepi
Executes up to the next machine instruction. The **stepi** command is different from the **nexti** command. The **stepi** command proceeds to the next instruction you can execute. As a result, if the current machine instruction contains a function call, execution stops on the first instruction of the called function. In contrast, if you enter the **nexti** command, execution does not stop within the called function.

stopi at *address*
stopi if *condition*
stopi at *address* if *condition*
Stops execution when one of the following conditions apply: the given condition is true, the given address is reached, the given subroutine or function is called, or the given variable is modified at an address. See the **stop** command description for information on how to specify **stopi** command arguments.

tracei [*address*] [if *condition*]
tracei [*variable*] [at *address*] [if *condition*]
Turns on tracing using a machine instruction address. If you enter the **tracei** command without arguments, the execution of the entire program is traced.

The variable condition is a Boolean expression that is evaluated prior to displaying the tracing information. If the expression is false, the information is not displayed. Section 3.2.3 describes the operators used in conditional expressions.

3.5 Sample Debugging Session

This section contains a sample debugging session for a VAX C program. Example 3–1 contains a listing file for a program that requires debugging. The program was compiled and linked without diagnostic messages. The error occurs in the program's calculations, not in its syntax.

The program was designed to generate a table of squares for all values between 1 and 10. However, each value is added to itself rather than multiplied by itself. This is an obvious error. For illustrative purposes, this section deals with the problem as if it were not obvious.

NOTE

This section does not address machine-level debugging techniques.

Example 3–1: Sample VAX C Program

```
.MAIN
V1.0

  1          #include "stdio.h"
113          #define STARTNUM 1
114          #define ENDNUM 10
115
116
117          main ()
118          {
119    1         int    Count;
120    1         long   Square;
121    1
122    1         printf ("Table of squares \n\n");
123    1
124    1         for (Count = STARTNUM; Count <= ENDNUM ; Count++){
125    2             Square = (long)Count + (long)Count;
126    2
127    2             printf ("Number: %d Square: %ld\n", Count, Square);
128    2             }
129    1         }
130
```

When you debug a program, you are trying to find out what is happening at key points in your program. To do this, you must be able to stop execution and examine program locations. The point at which you stop execution is called a breakpoint. Breakpoints are set with the **stop** command.

Use the **print** or **dump** commands to examine the contents of a location. After encountering a breakpoint, use the **cont**, **next**, **step**, **run** or **rerun** commands to resume program execution.

Example 3–2 shows a dialog for a terminal debugging session. Red print in the example indicates your input. The numbers are keyed to notes that explain the procedure.

Example 3–2: Sample Debugging Session

```
%❶ vcc -g square.c
%❷ dbx a.out
dbx version 2.0 of 10/24/86 5:11.
type 'help' for help.
reading symbolic information...
[using memory image in core]
(dbx)❸ func main
(dbx)❹ list
8       {
9           int    Count;
10          long   Square;
11
12          printf ("Table of squares \n\n");
13
14          for (Count = STARTNUM; Count <= ENDNUM ; Count++){
15              Square = (long)Count + (long)Count;
16
17              printf ("Number: %d Square: %ld\n", Count, Square);
(dbx)❺ stop at 17
[1] stop at "square.c":17
(dbx)❻ run
❼ Table of squares
❽ [1] stopped in main at line 17 in file "square.c"
   17          printf ("Number: %d Square: %ld\n", Count, Square);
(dbx)❾ print SQUARE
"SQUARE" is not defined
(dbx)❿ print Square
2
(dbx)print Count
1
(dbx)cont
Number: 1         Square: 2
[1] stopped in main at line 17 in file "square.c"
   17          printf ("Number: %d Square: %ld\n", Count, Square);
(dbx)print Square
4
(dbx)print Count
2
(dbx)cont
Number: 2         Square: 4
[1] stopped in main at line 17 in file "square.c"
   17          printf ("Number: %d Square: %ld\n", Count, Square);
(dbx)print Square
6
(dbx)print Count
3
(dbx)⓫ delete *
(dbx)⓬ cont
```

(continued on next page)

Example 3–2 (Cont.): Sample Debugging Session

```
Number: 3        Square: 6
Number: 4        Square: 8
Number: 5        Square: 10
Number: 6        Square: 12
Number: 7        Square: 14
Number: 8        Square: 16
Number: 9        Square: 18
Number: 10       Square: 20
execution completed
(dbx)❸ quit
%
```

Key to Example 3–2:

❶ Create an executable module with the default name a.out. This command line uses the **vcc** command program to invoke the VAX C compiler and the linker. The **–g** option generates the necessary dbx information.

❷ Invoke the dbx debugger, specifying the name of the executable module.

❸ Establish main as the current function.

❹ Display the function main on your terminal screen.

❺ Insert a breakpoint at line 17. Critical information is available at this point in the program.

❻ Initiate program execution.

❼ The program generates this message during execution.

❽ Execution halts when it reaches the breakpoint at line 17.

❾ An attempt is made to display the value of the variable SQUARE. This variable does not exist. The variable Square does exist.

❿ Display the contents of the variable Square.

⓫ Remove all breakpoints after isolating the cause of the error.

⓬ Continue program execution until processing is completed.

⓭ Exit from the dbx debugger.